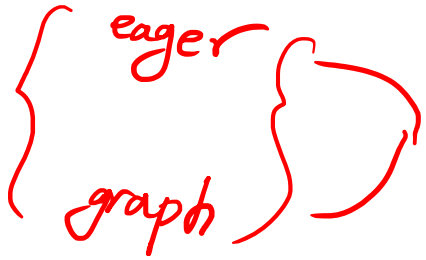
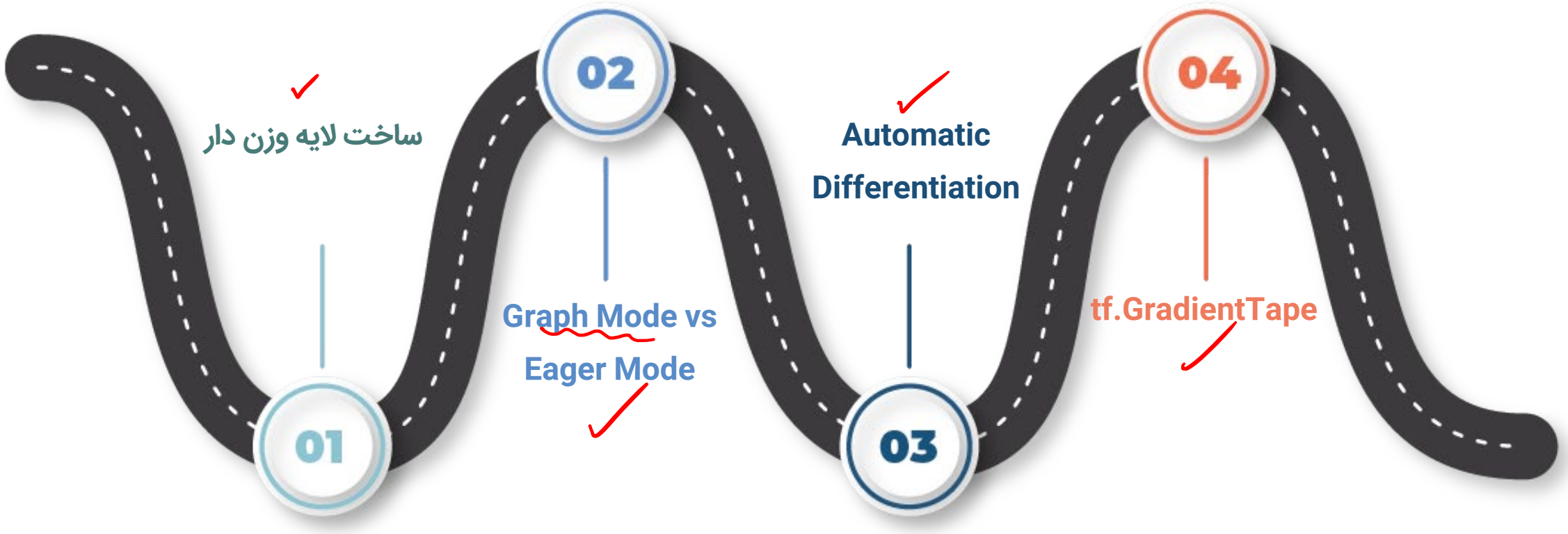
آکادمی ربوتک

دوره تنسورفلو پیشرفته



جلسه پنجم : ساخت لایه وزن دار و مدهای اجرایی Tensorflow

آنچه امروز خواهیم گفت :



انواع روش های نوشتن لایه در Tensorflow

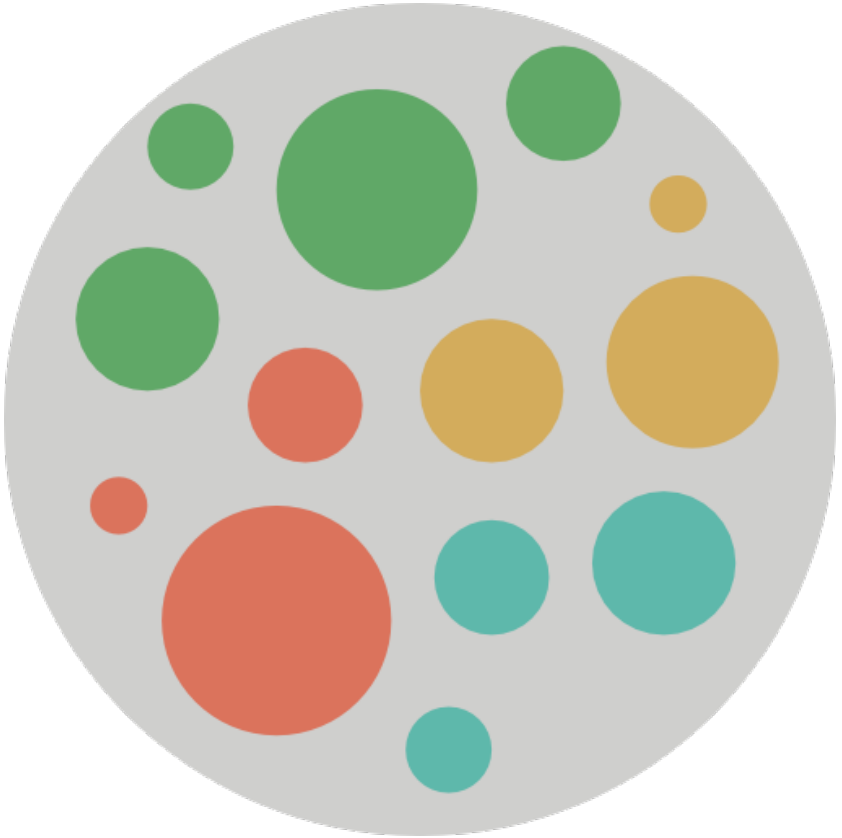
Dense

Trainable Layer

برای لایه های ساده

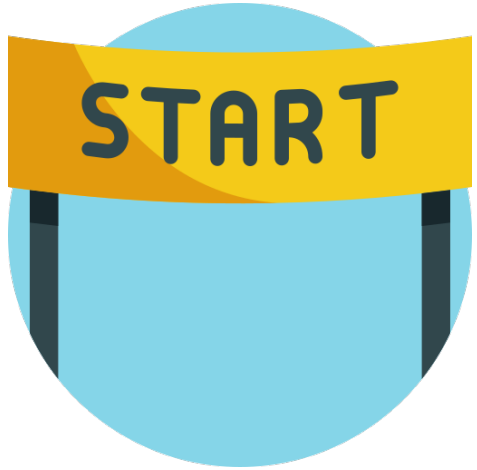
Lambda Layer

مثال : ساخت لایه Dense



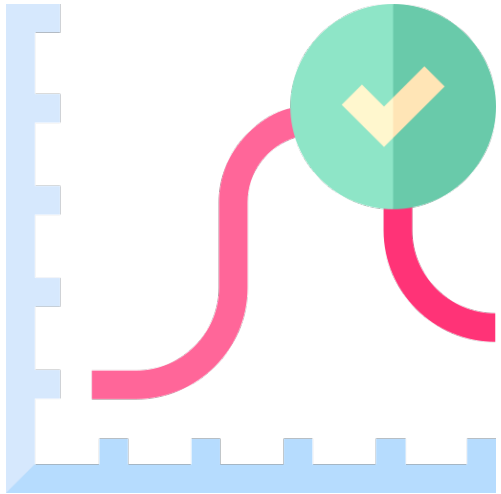
متد __init__

```
def __init__(self, units = 32):
    super().__init__()
    self.units = units
```



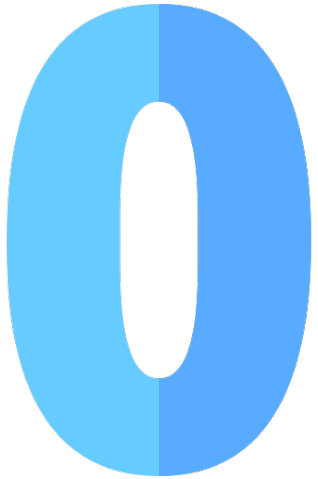
tf.random_normal_initalizer

```
obj=tf.random_normal_initializer(
        mean=0.0, stddev=0.05, seed=None)
```



tf.zeros_initalizer

```
tf.zeros_initializer()
```



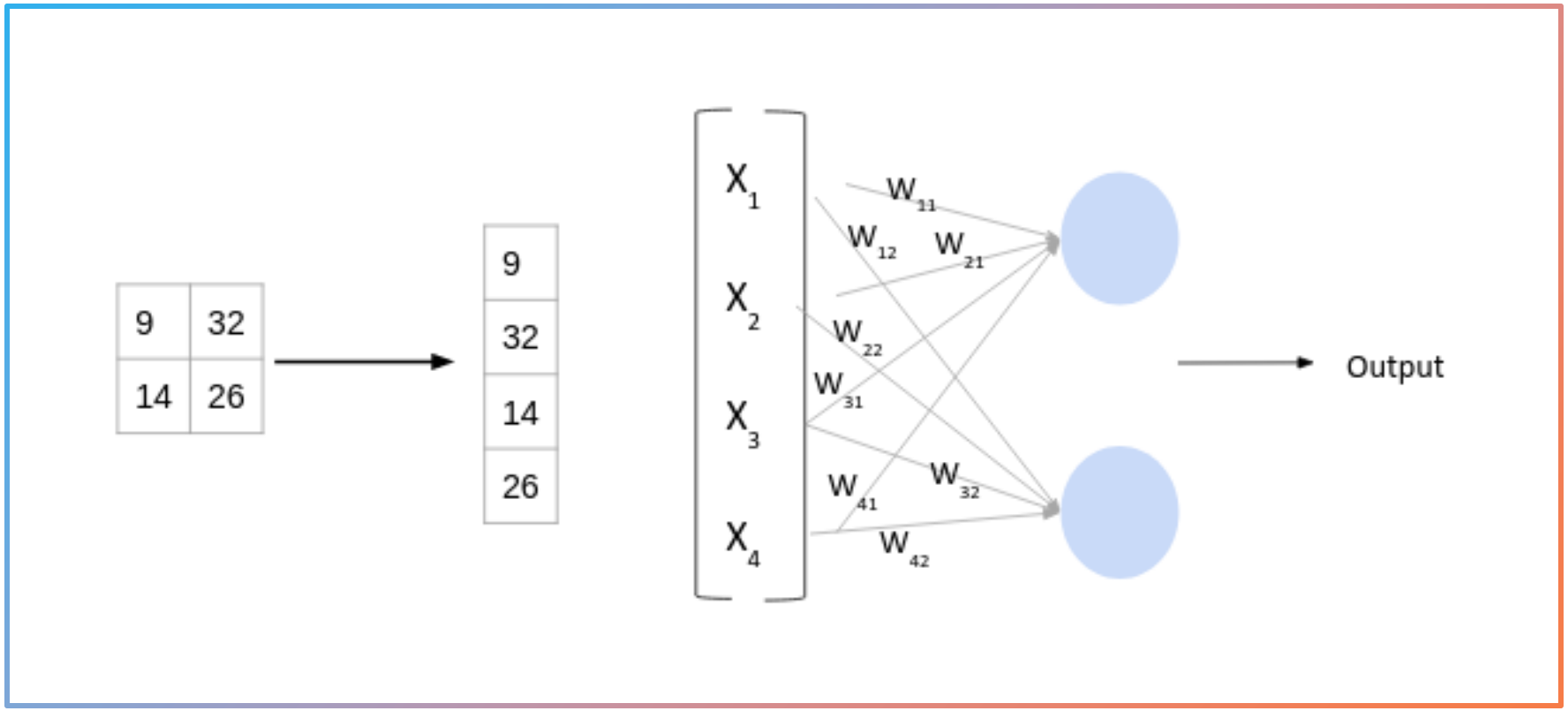
tf.Variable

```
tf.Variable(initial_value= ...,
            trainable=...,
            name = ... )
```

مثال tf.Variable

```
self.w = tf.Variable(initial_value=w_init(shape = (input_shape[-1], self.units),
                     dtype = "float32") ,trainable = True, name="kernel")
```

مروری بر ابعاد در لایه Dense



ابعاد ورودی و خروجی و وزن و بایاس به چه شکل است ؟

ابعاد محاسبات در Dense

$$y = x_{(1\times n)} W_{(n\times m)} + b_{(1\times m)}$$

input_shape[-1]

self.units

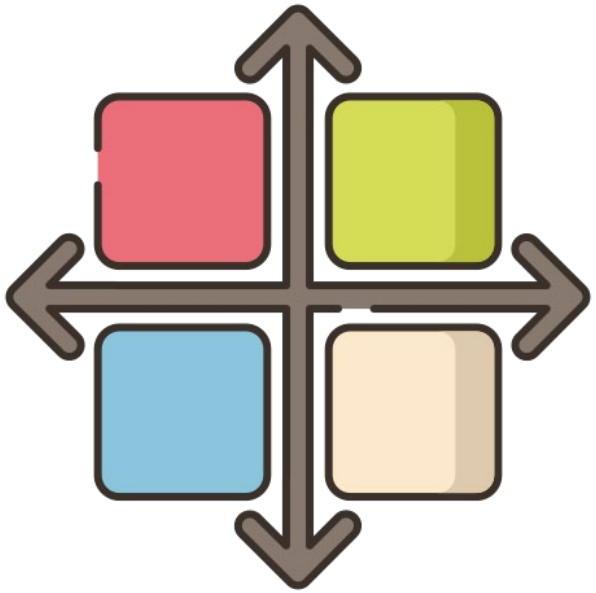
n : Input dim m: units

متد build

```
def build(self, input_shape):
  w_init = tf.random_normal_initializer()
    self.w = tf.Variable(initial_value=w_init(shape = (input_shape[-1], self.units),
                                dtype = "float32") ,trainable = True, name = "kernel")

   b_init = tf.zeros_initializer()
   self.b = tf.Variable(initial_value=w_init(shape=(self.units,),
                                dtype="float32"),  trainable =True, name="bias")
```

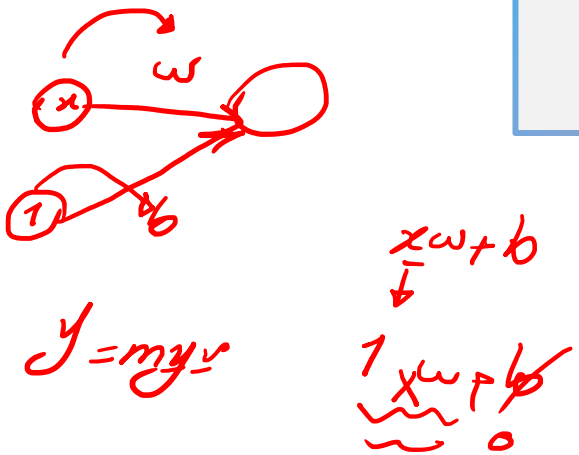
متد call

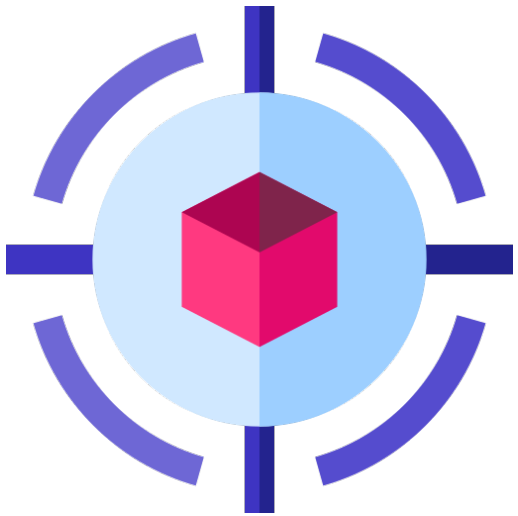
```
def call(self, inputs):
    return tf.matmul(inputs, self.w) + self.b
```



یک بررسی کوچک برای درک بهتر

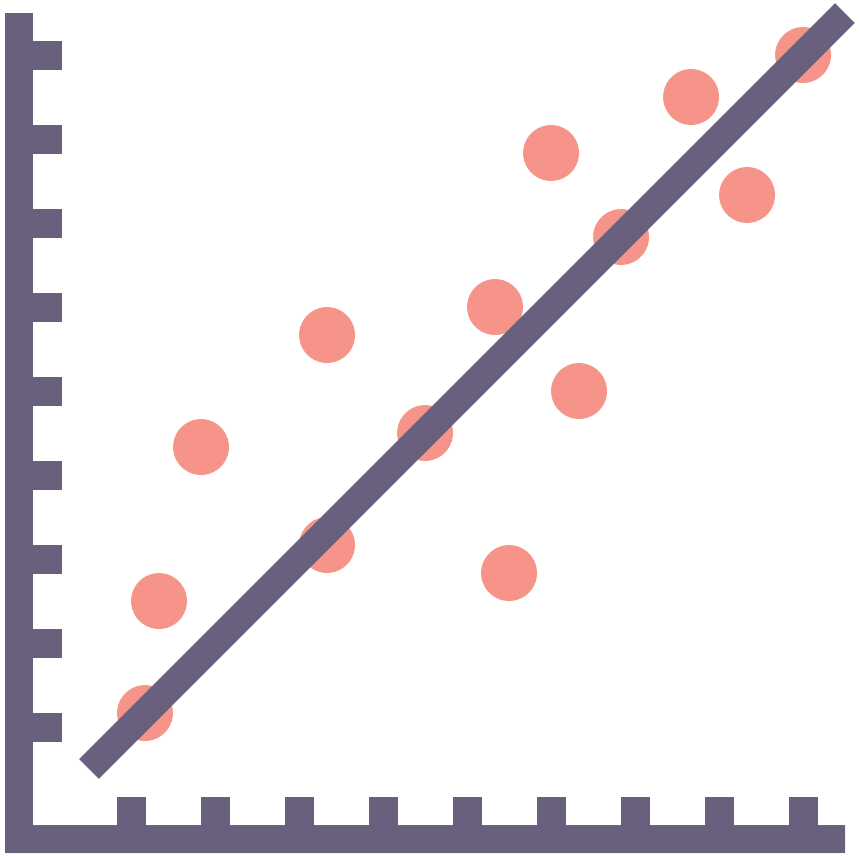
```
my_dense = SimpleDense(units=1)
x = tf.ones((1,1))
y = my_dense(x)
print(my_dense.variables)
```



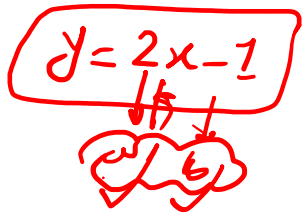


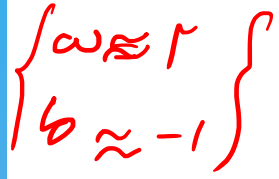
$$y = x_{(1\times n)}W_{(n\times m)} + b_{(1\times m)}$$

حل یک مثال:

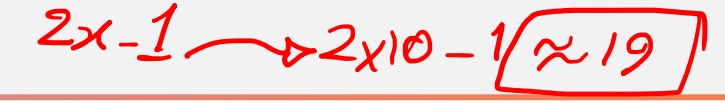


یک مثال ساده:



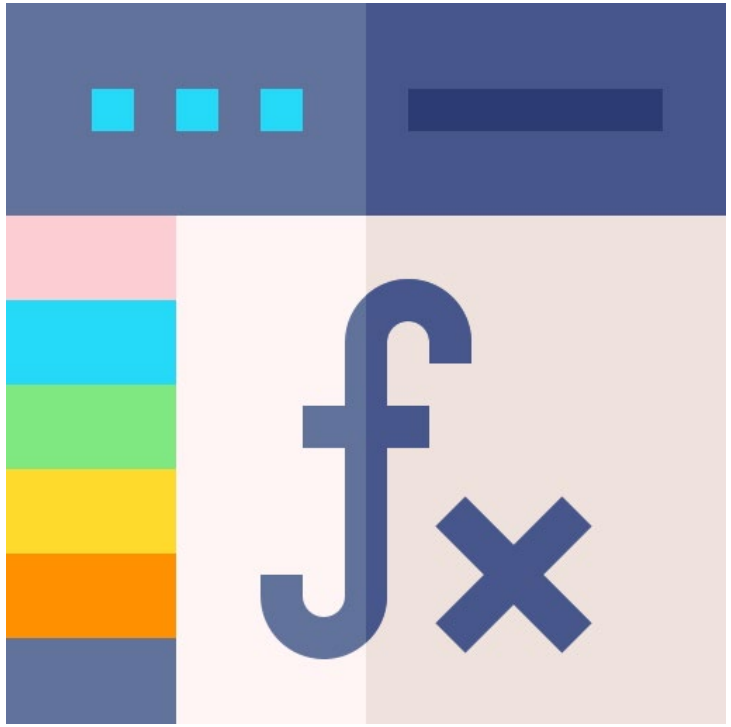



```
xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype = float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model = tf.keras.Sequential([
                            SimpleDense(1)
                            ])

model.compile(optimizer="sgd", loss = "mean_squared_error")
model.fit(xs, ys, epochs = 500)
print(model.predict([10.0]))
```



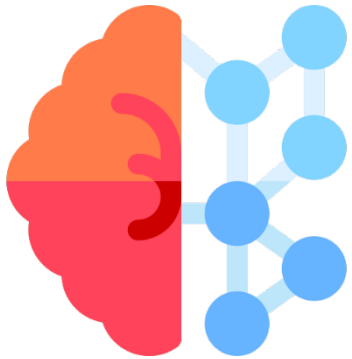
و البته ضرایب را هم ببینیم:

```
print(model.variables)
```

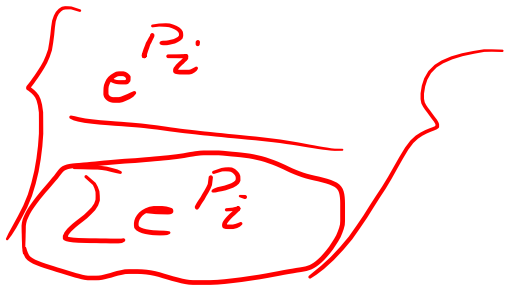


یک مثال کاملتر:

```
net = models.Sequential([
                        layers.Flatten(input_shape= (28 ,28)),
                        SimpleDense(units = 128),
                        layers.Lambda(lambda x:tf.maximum(x, 0.0)),
                        SimpleDense(units=10),
                        layers.Activation("softmax")
                        ])
```

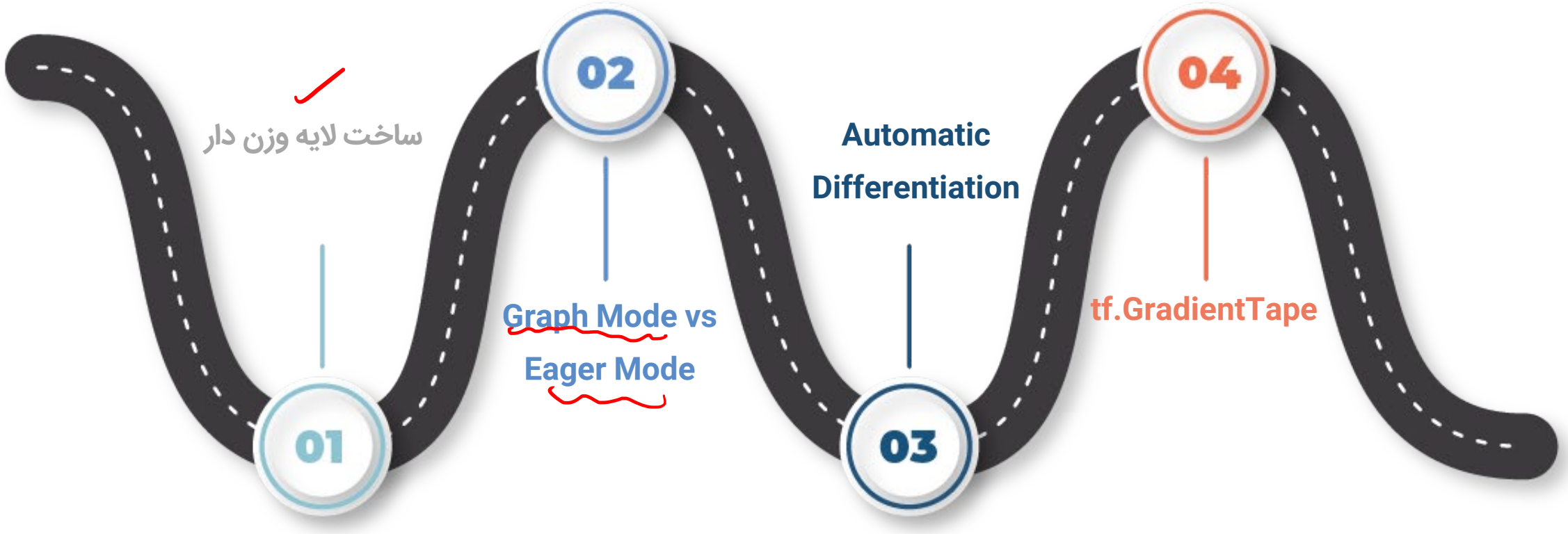


و حتی softmax

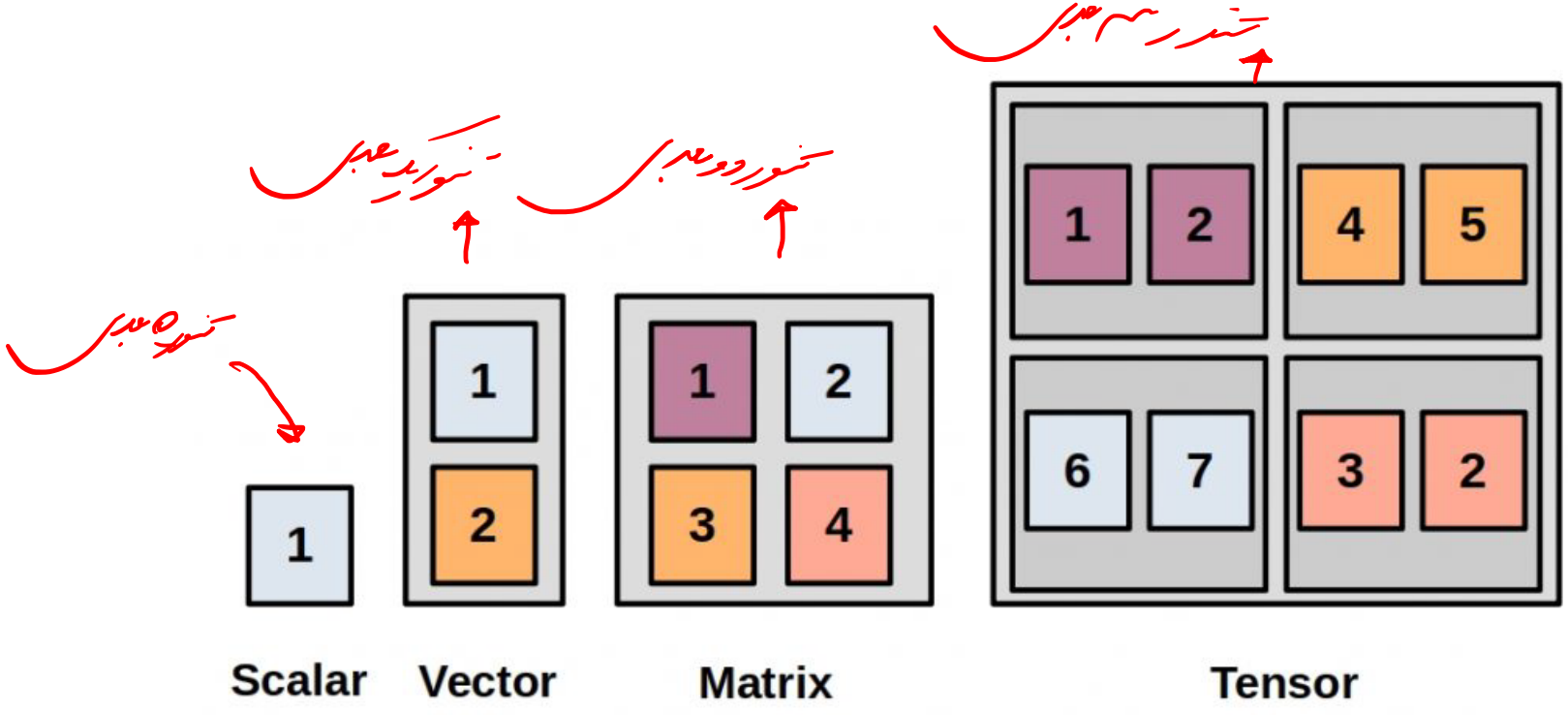


```
def softmax(x):
    e_x = tf.exp(x)
    return e_x/tf.reduce_sum(e_x)
```

آنچه امروز خواهیم گفت :

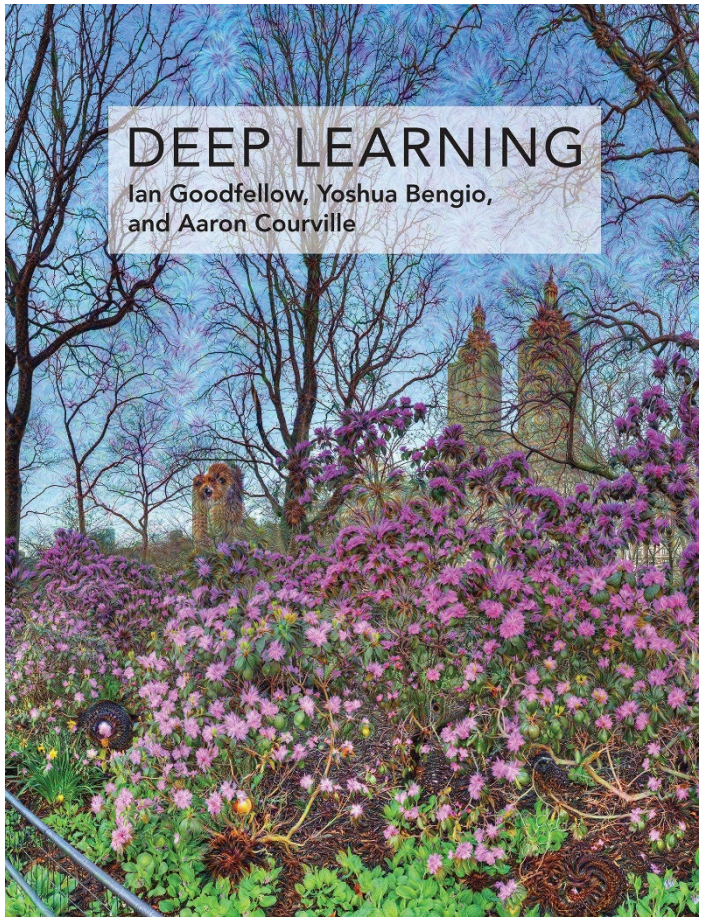


تنسور چیست ؟



تنسور یک داده ساختار انعطاف پذیر است که می تواند داده ها را در تعداد ابعاد متنوعی در خود نگه دارد.

تنسور از نگاه بزرگان



In the general case, an array of numbers arranged on a regular grid with a variable number of axes is known as a tensor.

دو روش ایجاد تنسور در تنسورفلو

Variables : **tf.Variable**

```
ten1 = tf.Variable("Hello", dtype = tf.string)
```

Constants : **tf.constant**

```
ten2 = tf.constant([1, 2, 3, 4, 5, 6])
```

ساختار یک تنسور در تنسورفلو

مثال:

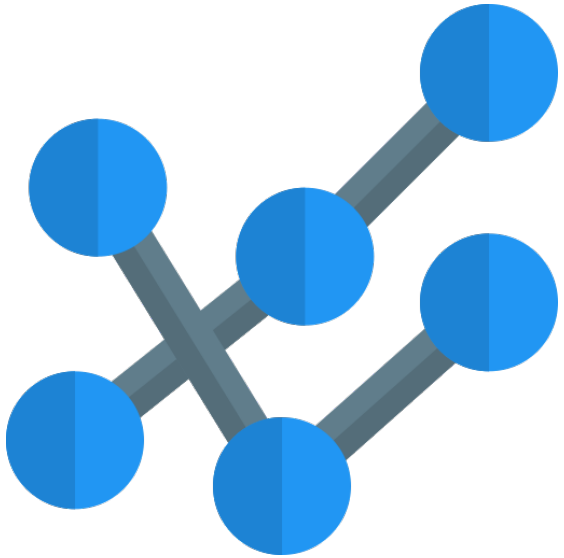
```
model = tf.keras.Sequential([
                              tf.keras.layers.Dense(1, input_shape=(1,))
                              ])
print(model.variables)
```

```
[ < tf.Variable 'dense/kernel:0' shape = (1, 1) dtype = float32, numpy = array([[1.02298
34]], dtype=float32) > ,
  < tf.Variable 'dense/bias:0' shape = (1,) dtype = float32, numpy = array([0.], dtype=f
loat32) > ]
```

ساخت Tensor با tf.Variable

```
vector = tf.Variable(initial_value=[1, 2])
```

```
<tf.Variable 'Variable:0' shape=(2,) dtype=int32, numpy=array([1, 2])>
```



یک تذکر در ساخت Tensor !

```
vector = tf.Variable([1, 2], tf.float32)
```

```
<tf.Variable 'Variable:0' shape = (2,) dtype = int32, numpy = array([1, 2]) >
```

ساخت Tensor با tf.Variable

```
vector = tf.Variable([1, 2, 3, 4], dtype=tf.float32)
```

```
<...dtype = float32, numpy = array([1., 2., 3., 4.], dtype=float32) >
```

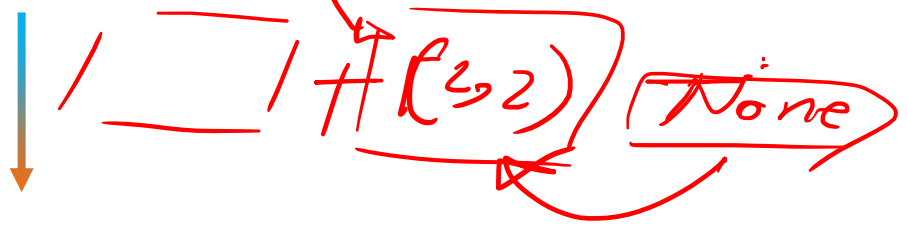
اشتباه متداول !

```
vector = tf.Variable([1, 2, 3, 4], shape=(2, 2))
```

```
ValueError: The initial value's shape((4,)) is not compatible with
 the explicitly supplied `shape` argument((2, 2)).
```

راه درست :

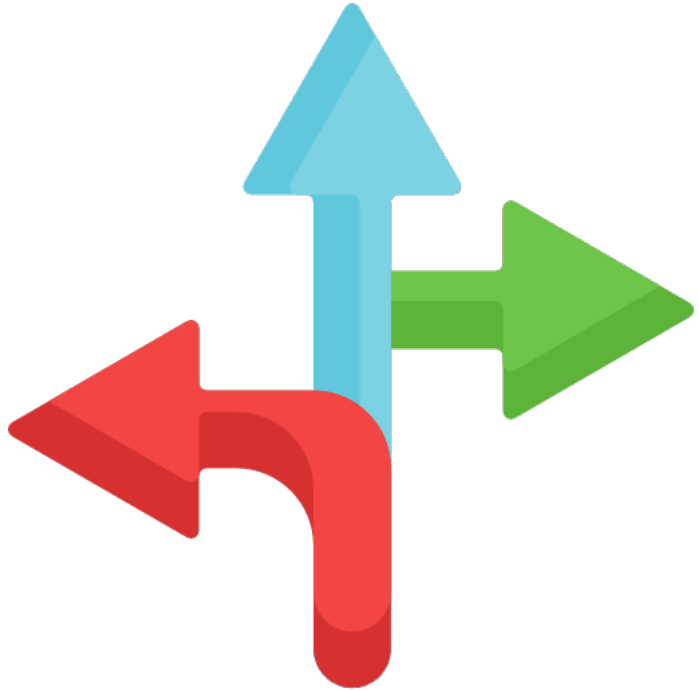
```
vector = tf.Variable([[1, 2],[3, 4]])
```




```
<tf.Variable 'Variable:0' shape = (2, 2) dtype = int32, numpy =
array([[1, 2],
       [3, 4]]) >
```

انعطاف پذیری tf.Variable در ساخت تنسور

```
str_tensor = tf.Variable("Elephant", dtype=tf.string)

complicated_number = tf.Variable(4+3j, dtype=tf.complex64)

first_primes = tf.Variable([1, 2, 3, 4])

linear_squares = tf.Variable([[1, 2], [3, 4]])
```



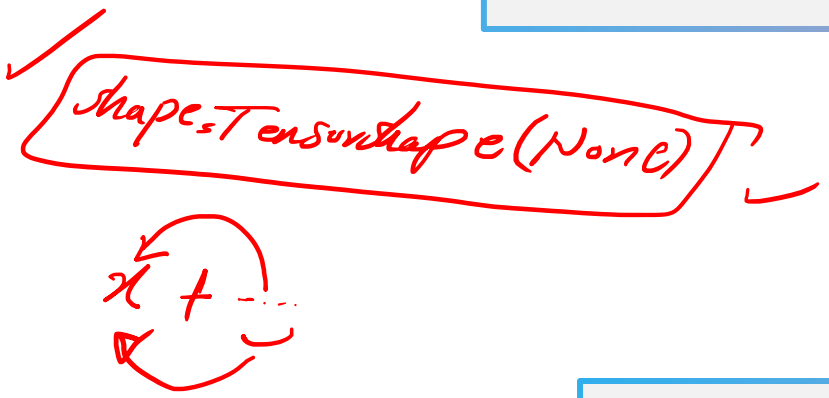
استفاده از tf.constant در ساخت Tensor

```
tensor = tf.constant([1, 2, 3])
```

```
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
```

اینجا از shape می توانیم استفاده کنیم !

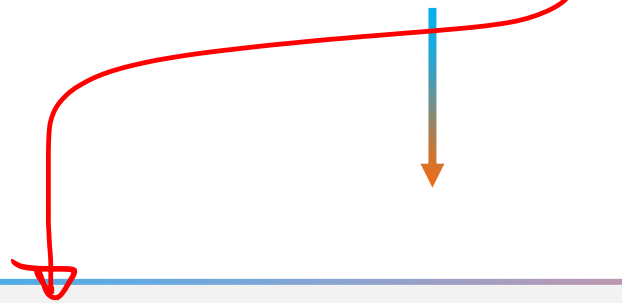
```
tensor = tf.constant([1, 2, 3, 4], shape=(2, 2))
```




```
tf.Tensor(
    [[1 2]
     [3 4]], shape=(2, 2), dtype=int32)
```

ساخت Tensor با المان های یکسان!

```
tensor = tf.constant(-1, shape=(2, 3))
```



```
tf.Tensor(
    [[-1 - 1 - 1]
     [-1 - 1 - 1]], shape=(2, 3), dtype=int32)
```

عملیات های ریاضی با تنسورها

عملیات های ریاضی بر روی تنسورها

```
tf.subtract([1, 2], [4, 3])
```
→ `[4 6]`

```
add_tensor = tf.add([1, 2], [3, 4])
```
→ `[-3 - 1]`

```
x = tf.constant([1, 2, 3, 4])
tf.math.multiply(x, x)
```
→ `[ 1, 4, 9, 16]`

تبدیل نوع داده تنسورها:

```
x = tf.Variable([1, 2, 3, 4])
x = tf.cast(x, tf.float32)
```

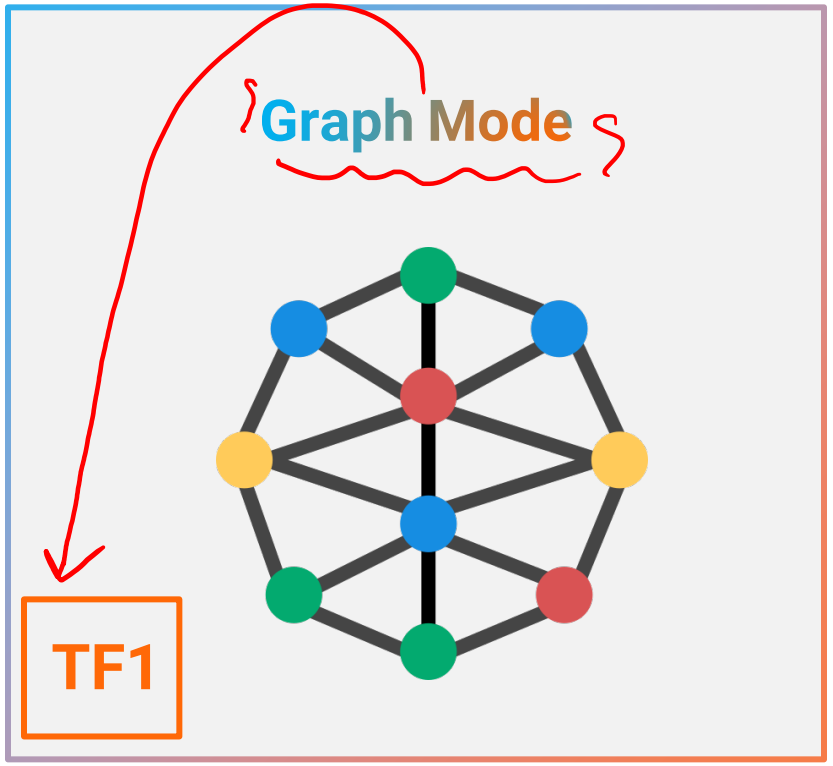
```
tf.Tensor([1. 2. 3. 4.], shape=(4,), dtype=float32)
```

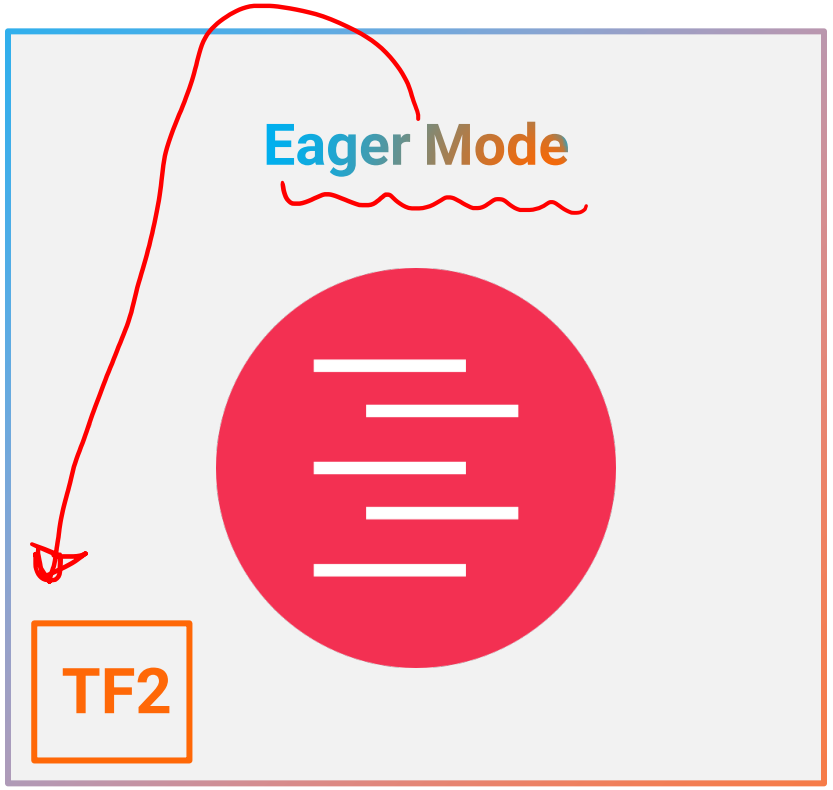
تذکر :

در انجام عملیات های تنسوری به یکی بودن نوع داده ها دقت کنید.

یه دور سریع همین دستورات رو اجرا کنیم !

مدهای اجرایی کد در Tensorflow

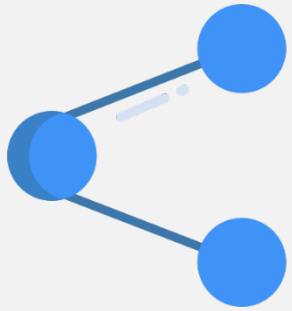




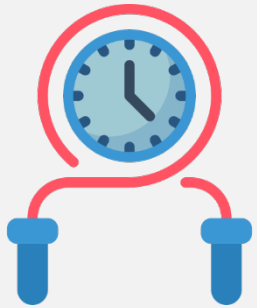
TF1

Graph Mode چگونه کار می کرد ؟

یک Computational Graph بساز



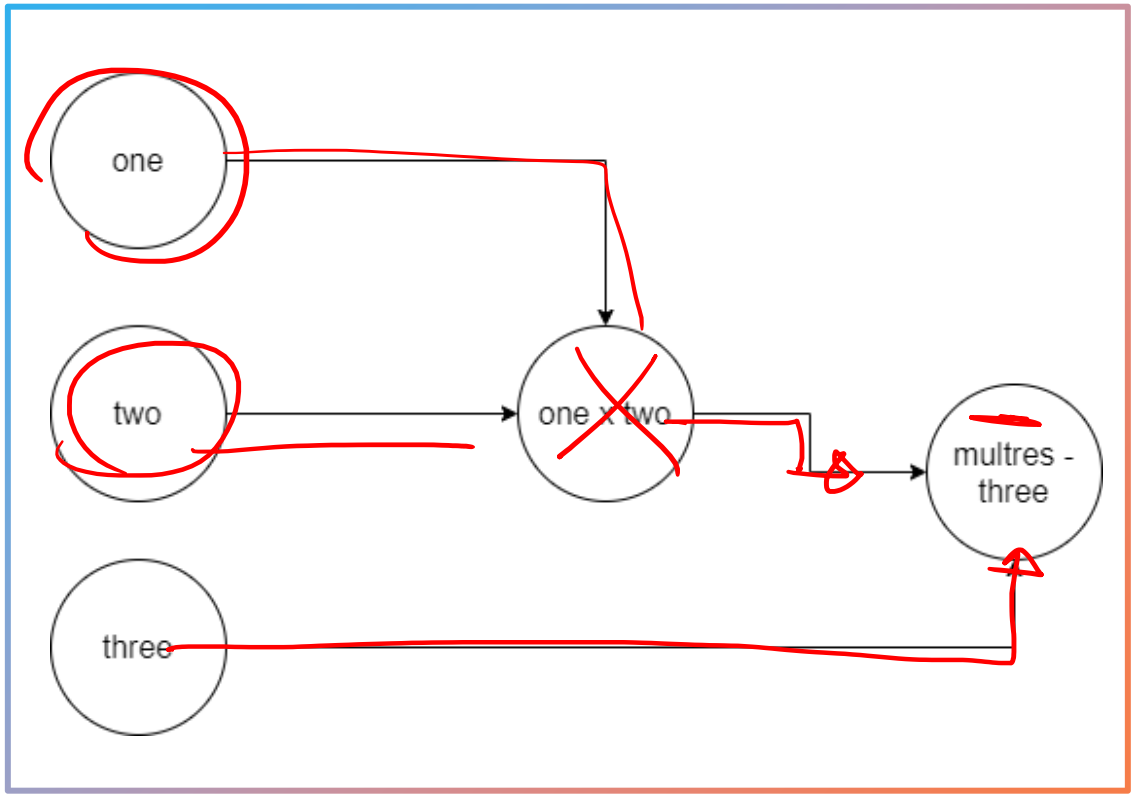
با session اجرا کن !



TF1

یک مثال برای درک بهتر:

```
one = tf.constant([12])
two = tf.constant([3])
three = tf.constant([2])
multres = tf.math.multiply(one, two)
subres = multres - three
```



TF1

یعنی خروجی این کد به شکل زیر بود:

```
one = tf.constant([12])
two = tf.constant([3])
three = tf.constant([2])
multres = tf.math.multiply(one, two)
subres = multres - three
```

12x3=36

36-2=34

```
Tensor("sub: 0", shape=(1,), dtype=int32)
```

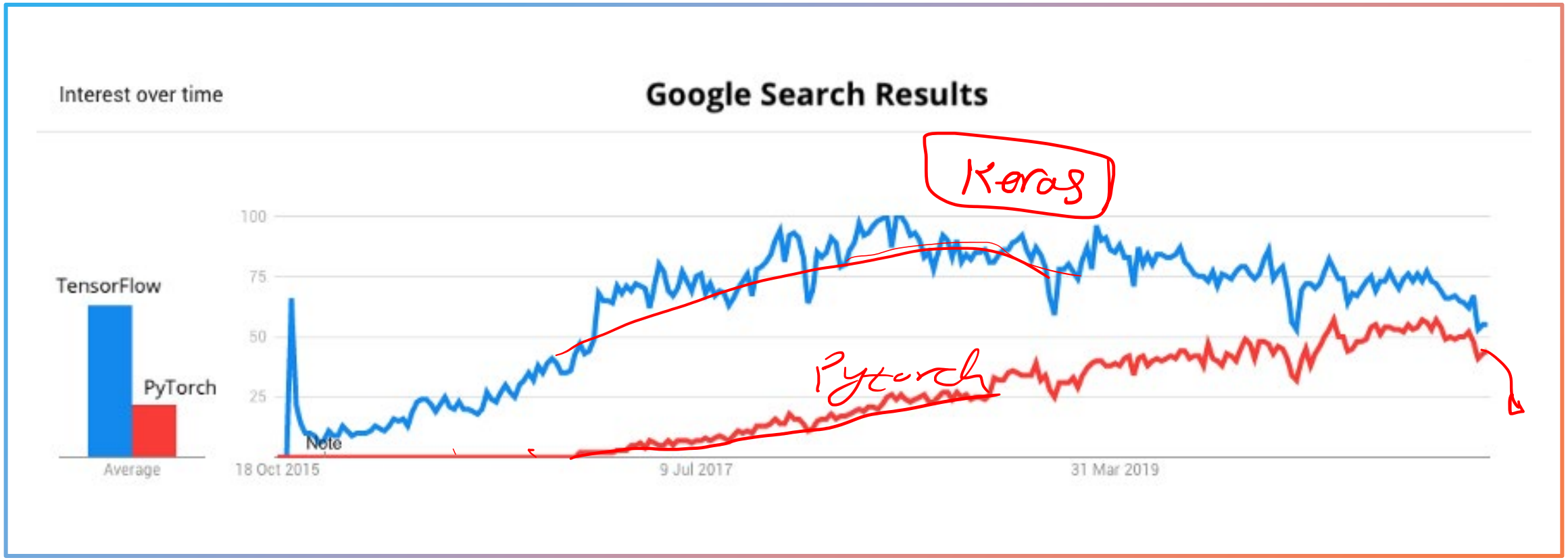
TF1

برای اجرا باید به صورت زیر عمل می کردیم :

```
with tf.Session() as sess:
    print(sess.run(c))
```

```
tf.Tensor([34], shape=(1,), dtype=int32)
```

تنسورفلو احساس خطر کرد !



و از Tensorflow2.0 مد Eager را فعال کرد.

دیگه نمیخواد گراف بسازی ! هر عملیاتی انجام میشه همونجا محاسبه کن !

```
one = tf.constant([12])
two = tf.constant([3])
three = tf.constant([2])
multres = tf.math.multiply(one, two)
subres = multres - three
```

```
tf.Tensor([34], shape=(1,), dtype=int32)
```

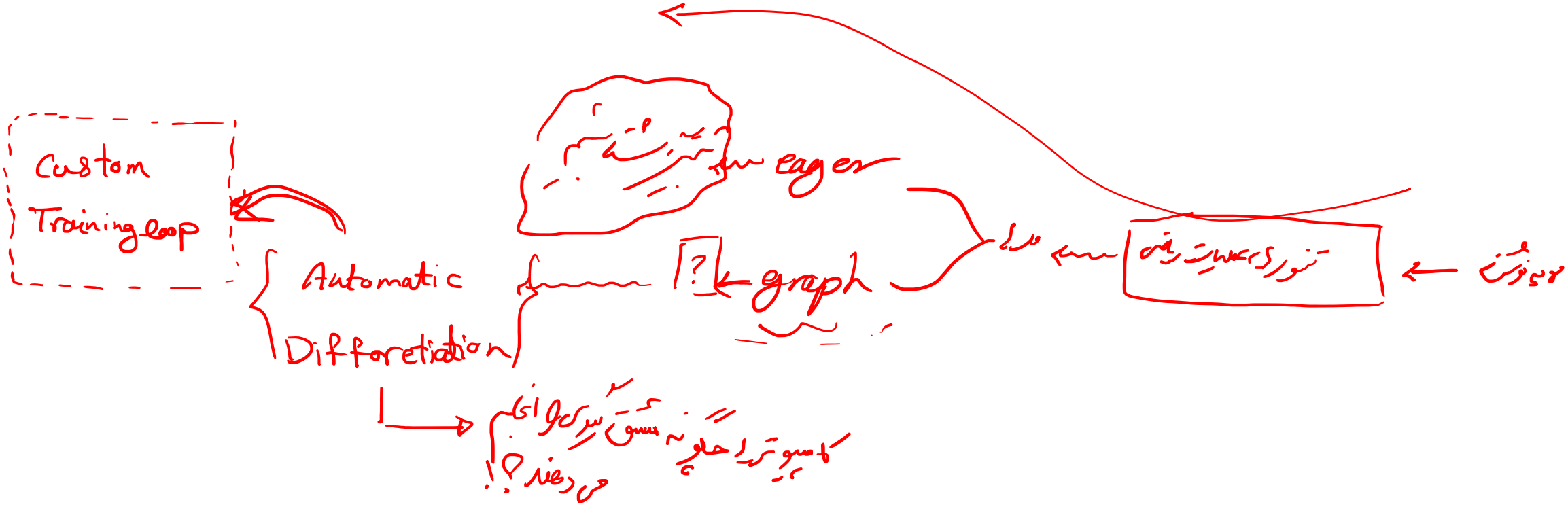
دو تا دستور دست گرمی :

```
tf.compat.v1.disable_eager_execution()
```

غیر فعال کردن eager mode

```
print(tf.executing_eagerly())
```

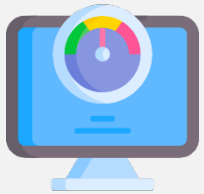
چک کردن فعال بودن eager mode


Custom
Training loop
Automatic
Differentiation
eager
?
graph

Eager Mode را کمی بیشتر بشناسیم:

Evaluate Values immediately

Broadcast Support



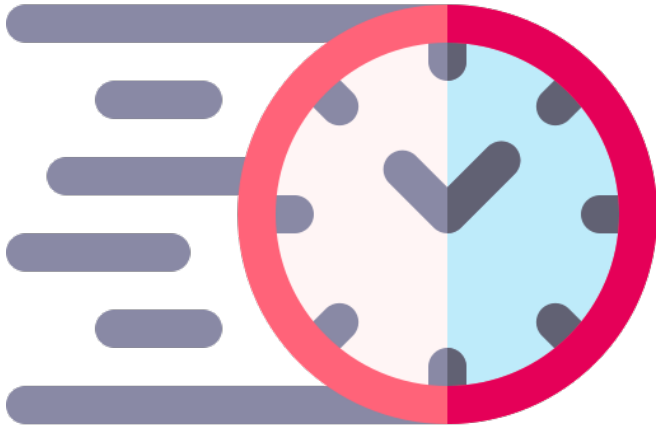
Operator Overloading

Numpy Compatibility

Evaluate Values immediately

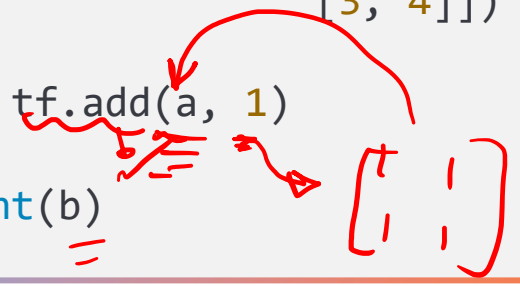
```
x = 2
x_squared = tf.square(x)
print("x_squared:", x_squared)
```

```
x_squared: tf.Tensor(4, shape=(), dtype=int32)
```

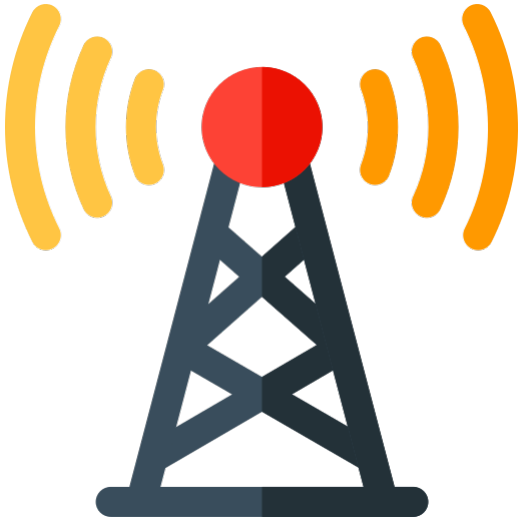


Broadcast Support

```
a = tf.constant([[1, 2],
                 [3, 4]])

b = tf.add(a, 1)

print(b)
```



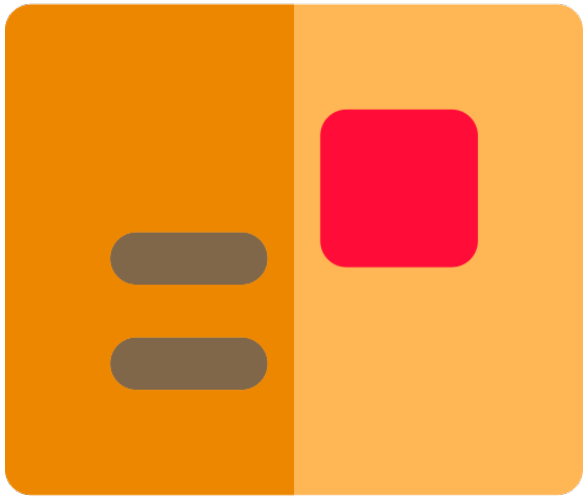
```
tf.Tensor(
    [[2 3]
     [4 5]], shape=(2, 2), dtype=int32)
```



Overload Operator

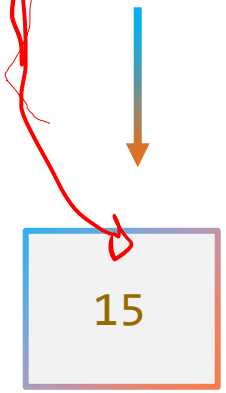
```
a = tf.constant([[1, 2],
                 [3, 4]])

b = a**2

print(b)
```

```
tf.Tensor(
    [[1  4]
     [9 16]], shape=(2, 2), dtype=int32)
```



Numpy Compatibility

```
a = tf.constant(5)
b = tf.constant(3)
c = np.multiply(a, b)
print(c)
```

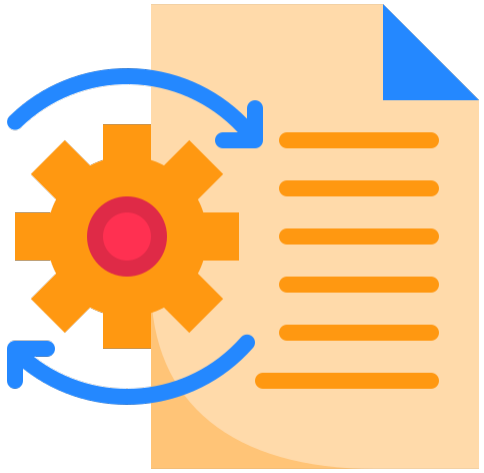


NumPy

Numpy Interoperability

```
ndarray = np.ones([3, 3])

tensor = tf.multiply(ndarray, 3)

print(tensor)
```

```
tf.Tensor(
    [[3. 3. 3.]
     [3. 3. 3.]
     [3. 3. 3.]], shape=(3, 3), dtype=float64)
```



تبدیل Tensor به Numpy

```
print(tensor.numpy())
```

```
[[3. 3. 3.]
 [3. 3. 3.]
 [3. 3. 3.]]
```

Evaluating Variable

```
v = tf.Variable(0.0)
print(v+1)
```

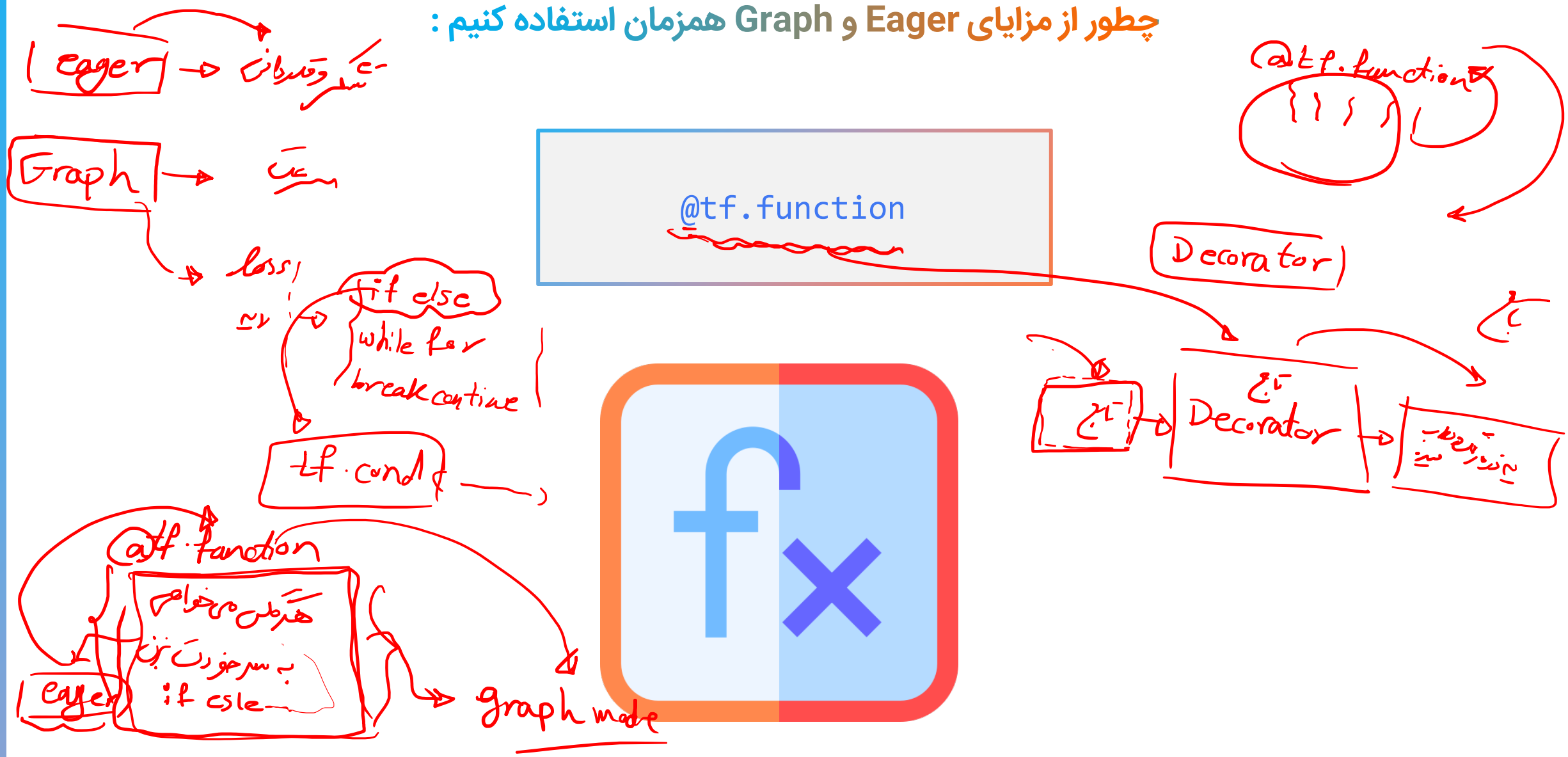
```
tf.Tensor(1.0, shape=(), dtype=float32)
```

Examine Custom Layers

```
class MyLayer(tf.keras.layers.Layer):


    def __init__(self) -> None:
        super().__init__()
        self.my_var = tf.Variable(100)
        self.my_other_var_list = [tf.Variable(x) for x in range(2)]


m = MyLayer()
out = [variable.numpy() for variable in m.variables]
```

```
[100, 0, 1]
```

چطور از مزایای Eager و Graph همزمان استفاده کنیم :



یک تابع عادی (در مد eager)

```
def eager_function(x):
  result = x ** 2
  print(result)
  return result

x = tf.constant([1.0, 2.0, 3.0, 4.0])

eager_function(x)
```

```
tf.Tensor([1, 4, 9, 16], shape=(4,), dtype=float32)
```

تبدیل همین تابع به Graph Mode

روش اول

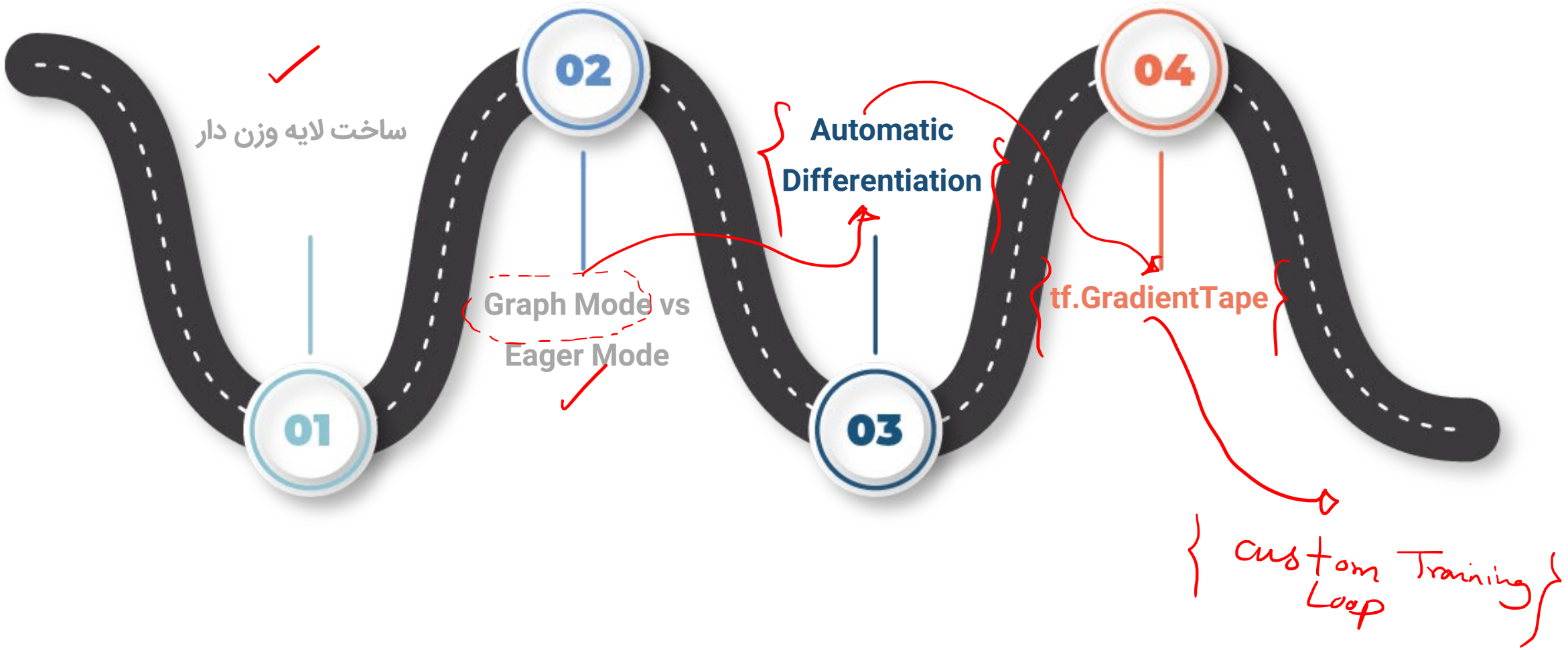
```
def eager_function(x):
  result = x ** 2
  print(result)

x = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0])

graph_function = tf.function(eager_function)
graph_function(x)
```

روش دوم

```
@tf.function
def eager_function(x):
  result = x ** 2
  print(result)

x = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0])

eager_function(x)
```

```
Tensor("pow:0", shape=(4,), dtype=float32)
```

آنچه تاکنون گفته ایم.



کامپیوترها چگونه مشتق میگیرند ؟

طوری که ما مشتق میگیریم :

$$y = e^{2x} - x^3 \longrightarrow y = 2e^{2x} - 3x^2$$

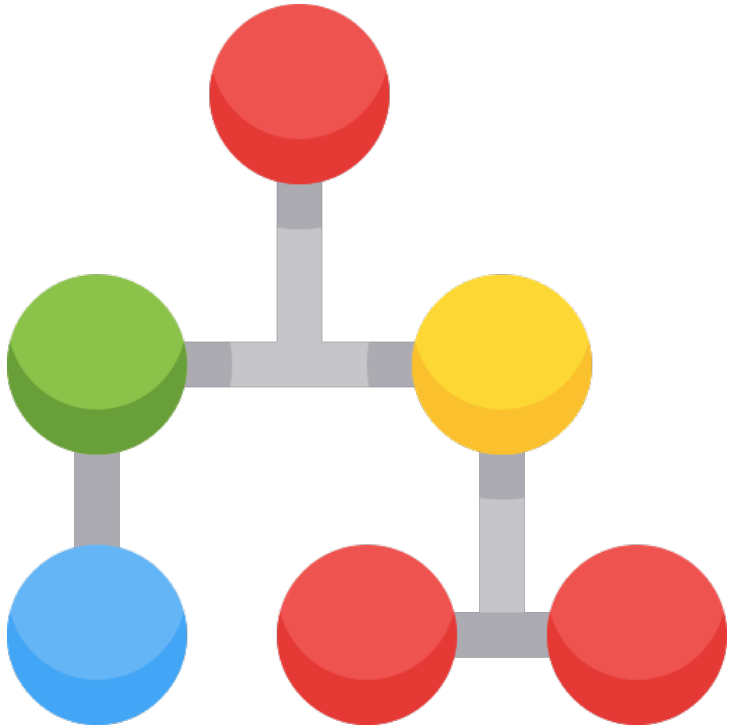
```
def f(x):
    return np.exp(2*x) - x**3
```

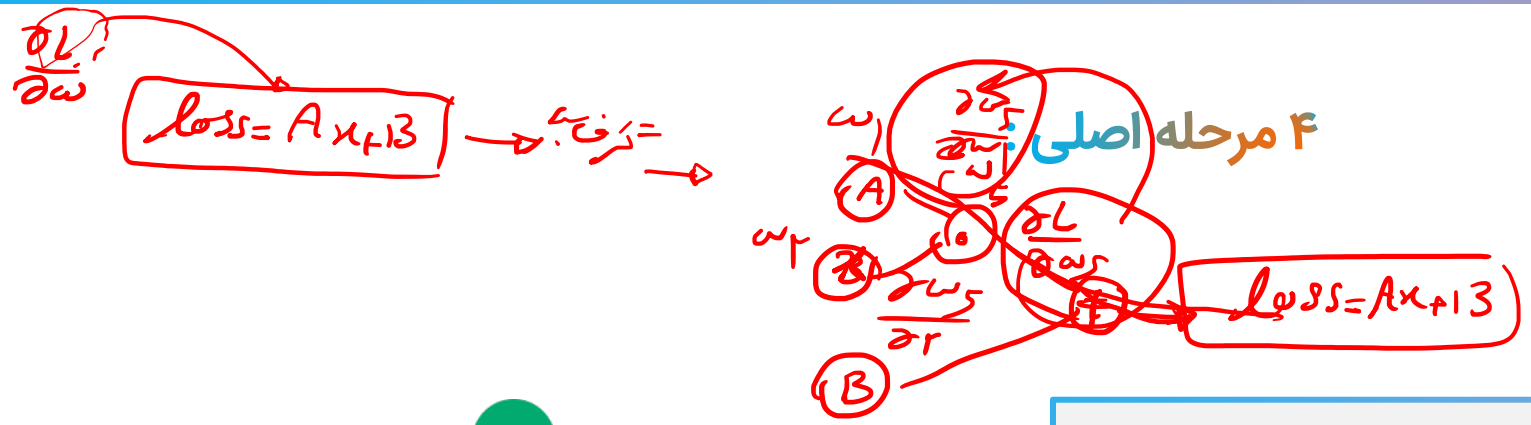
```
def f_prime(x):
    return 2*np.exp(2*x) -3*x**2
```

چطور کردی از این روش استفاده می‌شد

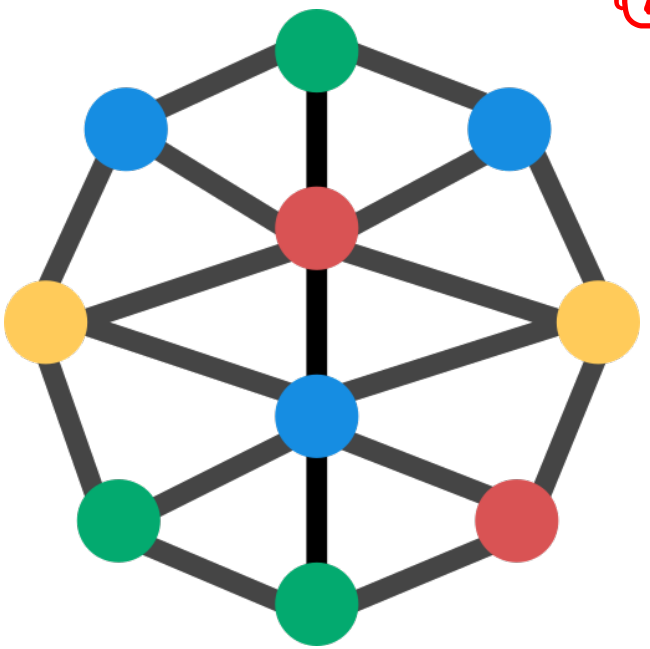
Automatic Differentiation

یکی از کاربردهای مهم Computational Graph در Automatic Differentiation است.





۴ مرحله اصلی



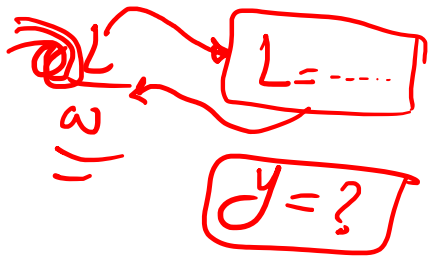
مرحله ۱ :از رابطه خروجی گراف بساز

مرحله ۲: مشتق گیری های جزیی را برای هر node انجام بده.

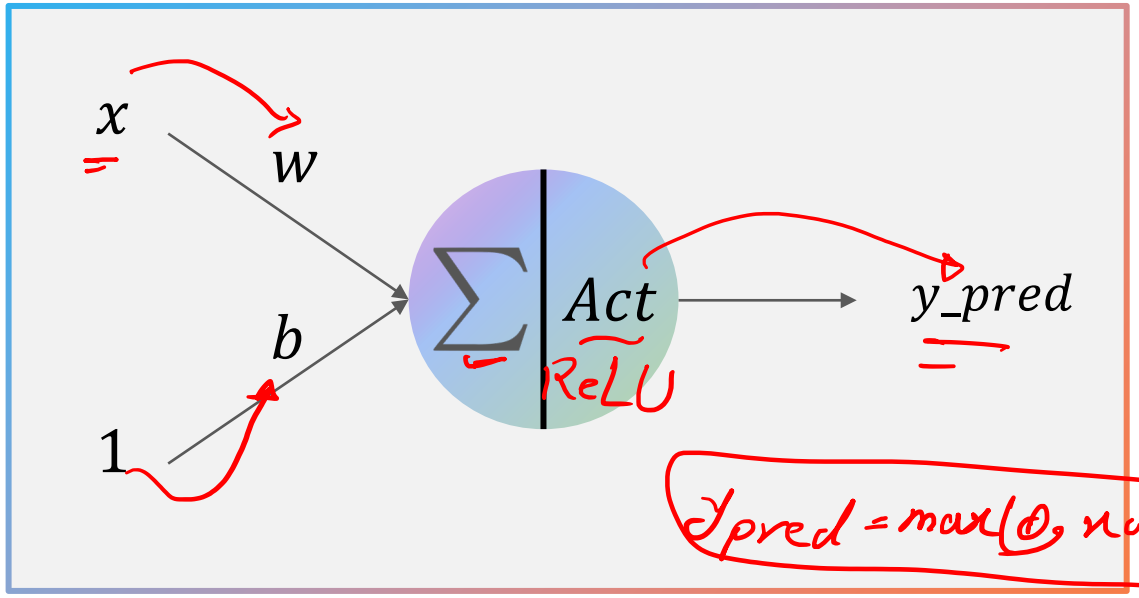
مرحله ۳: عملیات forward را بر روی گراف انجام بده.

مرحله ۴ : به کمک Backpropagation مشتق های نهایی را محاسبه کن.

یک مساله ساده در نظر بگیرید:



$$w^+ = w^- \frac{\partial L}{\partial w}$$

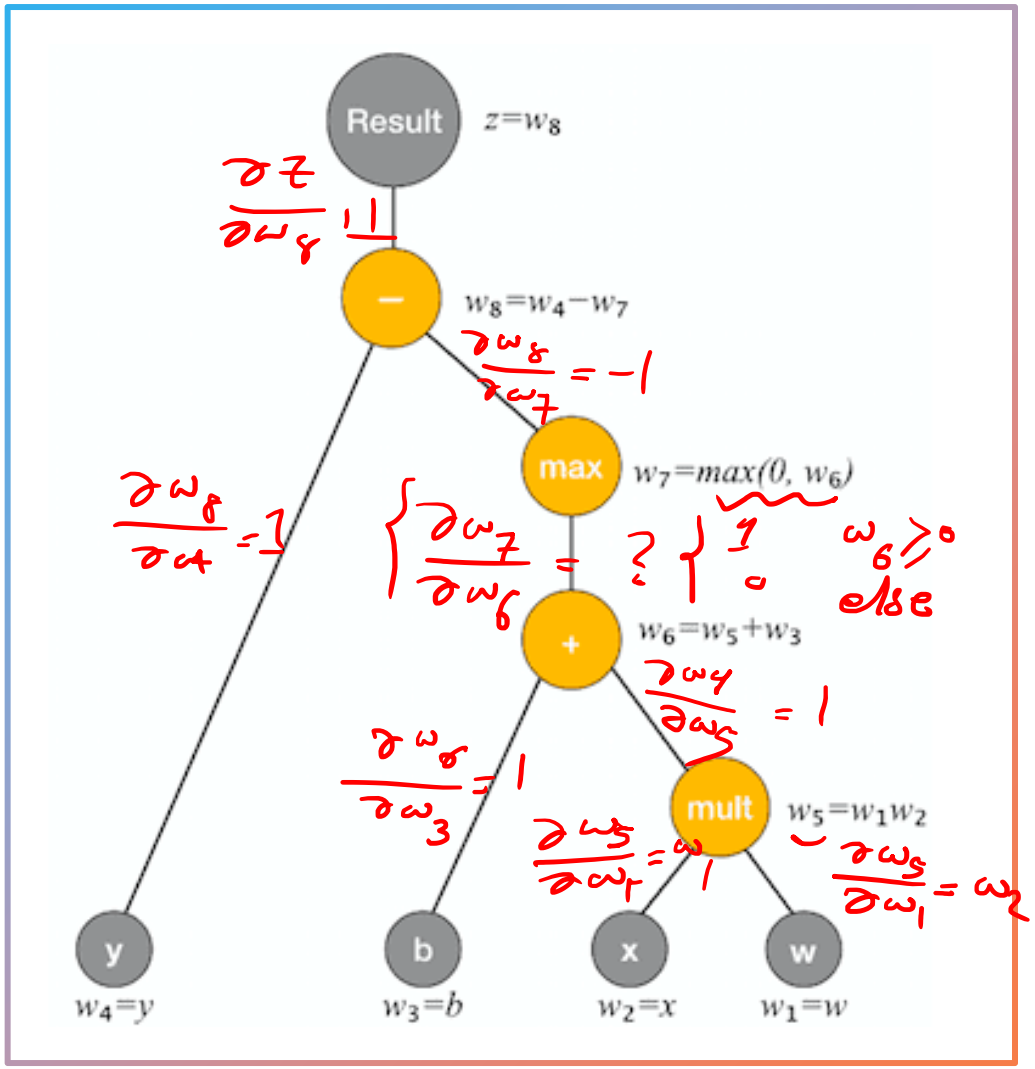


$$C(y, w, x, b) = y - max(0, w \cdot x + b)$$

ypred = max(0, wx+b)

loss=

مرحله ۱ : گراف ساخته می شود

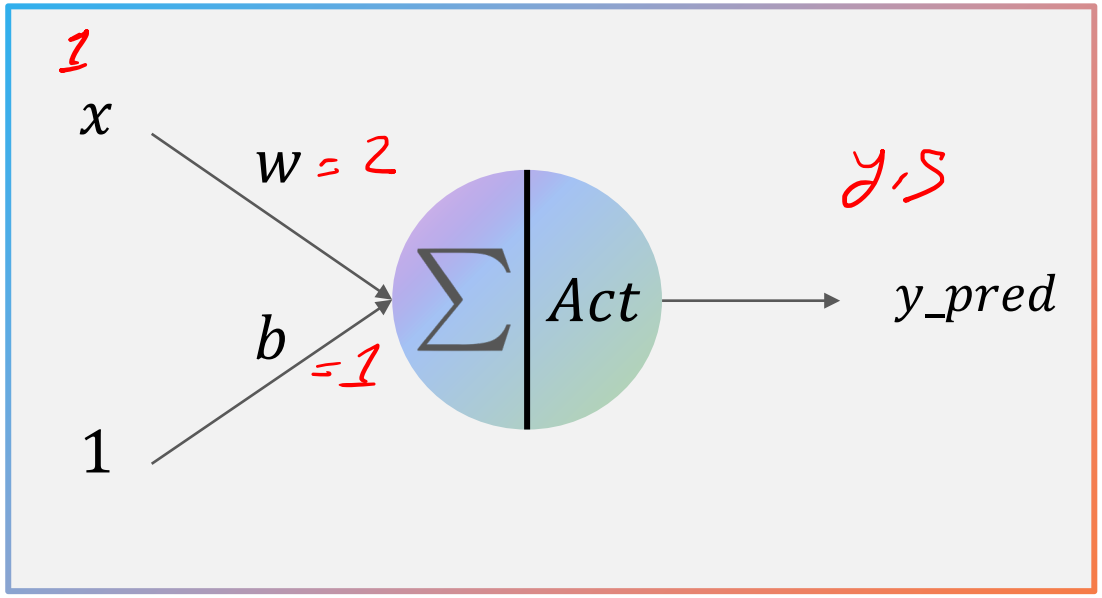


$$C(y, w, x, b) = y - max(0, w \cdot x + b)$$

مرحله ۲:مشتقات جزیی محاسبه می گردد.

$\frac{\delta w_5}{\delta w_1} = \frac{\delta w_1 w_2}{\delta w_1} = w_2$	$\frac{\delta w_5}{\delta w_2} = \frac{\delta w_1 w_2}{\delta w_2} = w_1$
$\frac{\delta w_6}{\delta w_5} = \frac{\delta w_5 + w_3}{\delta w_5} = 1$	$\frac{\delta w_6}{\delta w_3} = \frac{\delta w_5 + w_3}{\delta w_3} = 1$
$\frac{\delta w_7}{\delta w_6} = \frac{\delta \max(0, w_6)}{\delta w_6} = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$	$\frac{\delta w_8}{\delta w_7} = \frac{\delta w_4 - w_7}{\delta w_7} = -1$
$\frac{\delta w_8}{\delta w_4} = \frac{\delta w_4 - w_7}{\delta w_4} = 1$	$\frac{\delta z}{\delta w_8} = 1$

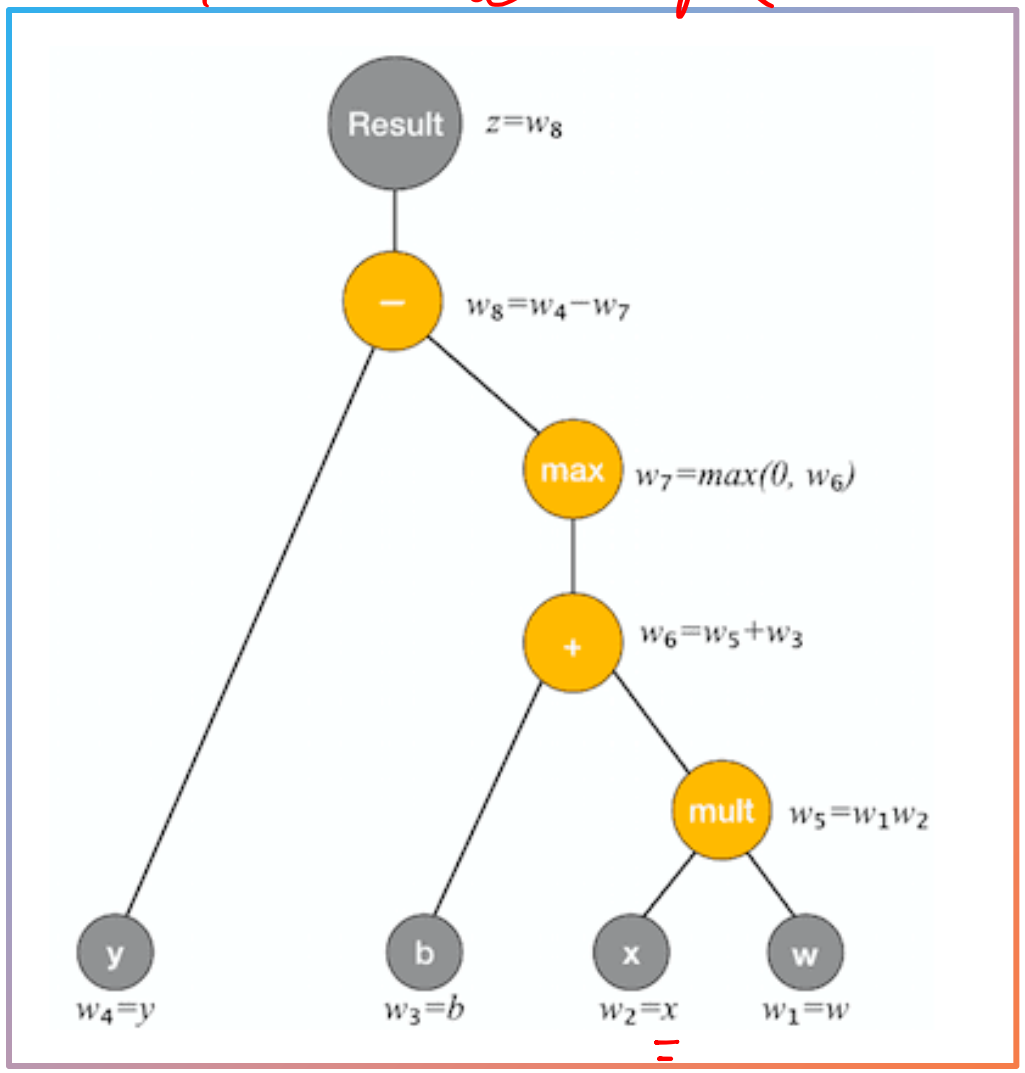
ورودی های فرضی :



$$y = 5, \quad w = 2, \quad x = 1, \quad b = 1$$

مرحله ۳ : خروجی ها محاسبه می گردد.

Computational Graph



Node	Expression	Value
w_1	w	2
w_2	x	1
w_3	b	1
w_4	y	5
w_5	$w_1 \cdot w_2$	2
w_6	$w_5 + w_3$	3
w_7	$\max(0, w_6)$	3
w_8	$w_4 - w_7$	2
z	w_8	2

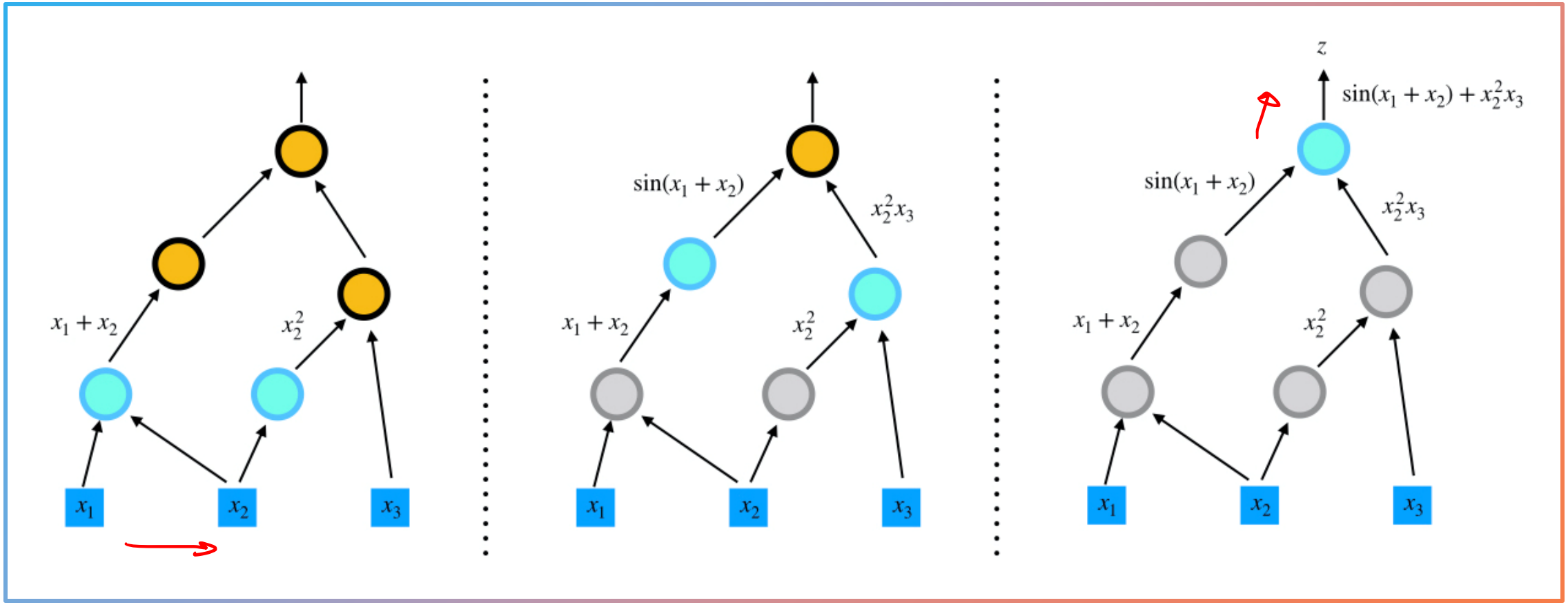
loss

مرحله ۴ : با BP مشتق نهایی محاسبه می گردد.

$$\frac{\delta z}{\delta w_1} = \frac{\delta z}{\delta w_8} \times \frac{\delta w_8}{\delta w_7} \times \frac{\delta w_7}{\delta w_6} \times \frac{\delta w_6}{\delta w_5} \times \frac{\delta w_5}{\delta w_1} = 1 \times (-1) \times \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases} \times 1 \times w_2 = \begin{cases} 0, & x \leq 0 \\ 1, & x > 0 \end{cases}$$

مثالی دیگر از ساخت یک گراف

$$z = \sin(x_1 + x_2) + x_2^2 x_3$$



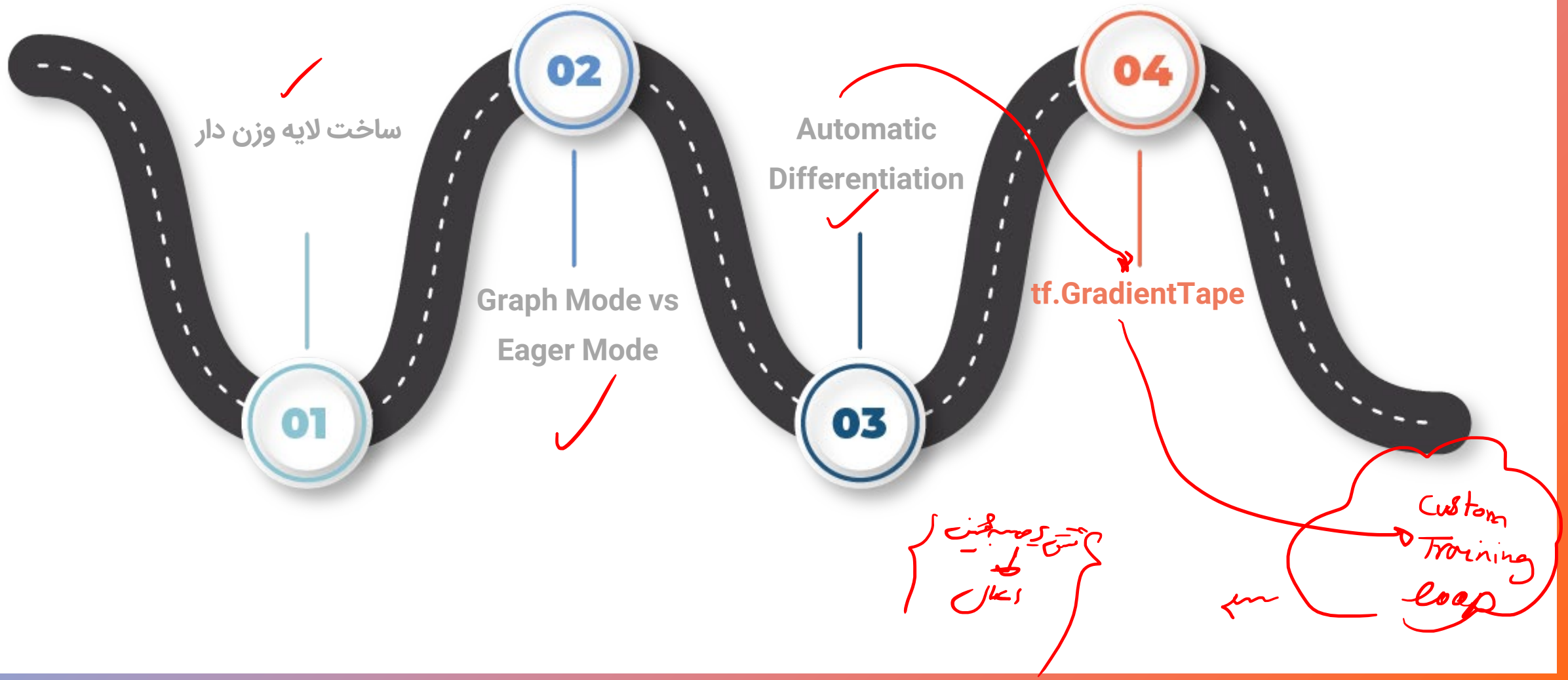
Pytorch و Tensorflow با این روش وزن ها را آپدیت می کنند.

Theano ,
JAX

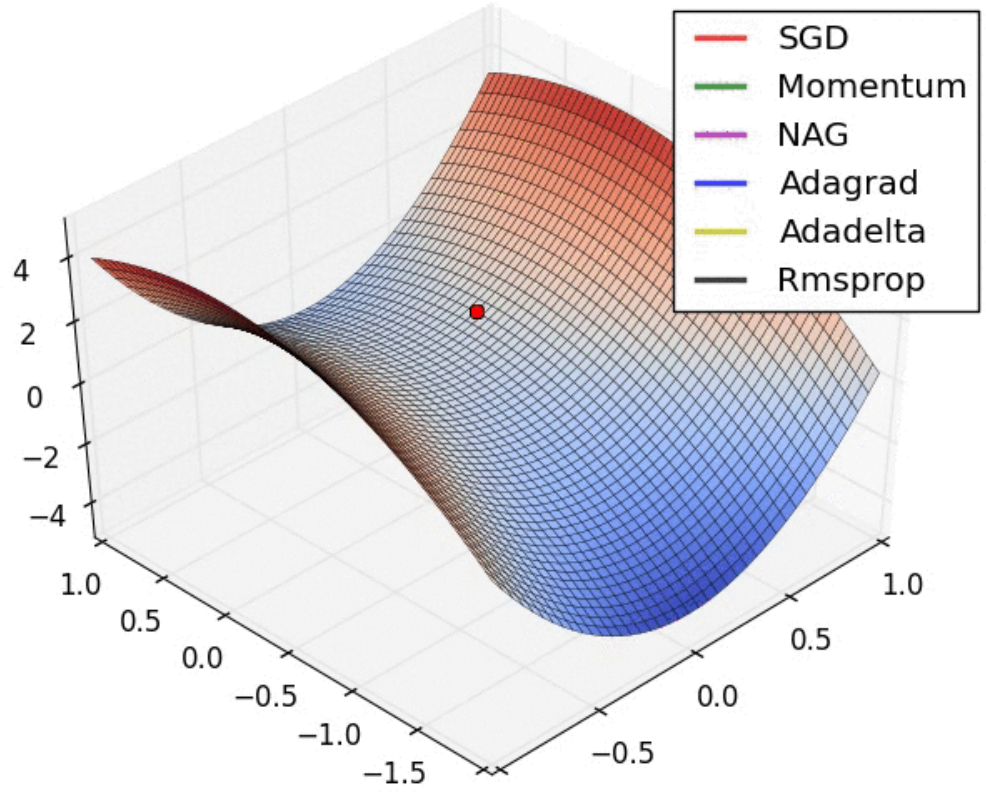
tf.GradientTape

torch.autograd

آنچه تاکنون گفته ایم.



tf.GradientTape



مثال ۱: نحوه استفاده از tf.GradientTape

```
x = tf.constant(5.0)
with tf.GradientTape() as tape:
    tape.watch(x)
    y = x**3

print(tape.gradient(y, x).numpy())
```

به کمک watch ثابت ها و متغیرها شروع به track کردن می شوند.

به کمک gradient مشتق گیری نهایی انجام می گیرد.

$$\frac{\partial y}{\partial x} = 3x^2 = 3 \times 5^2 = 125$$

مثال ۲: استفاده از GradientTape برای Variable ها

```
x = tf.Variable(6.0, trainable=True)
with tf.GradientTape() as tape:
    y = x**3

print(tape.gradient(y, x).numpy())
```

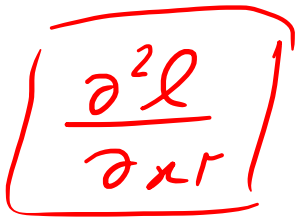
متغیرهای trainable به صورت پیش فرض Track می شوند.

مثال ۳ : متوقف کردن ردیابی اتوماتیک متغیرها توسط GradientTape

```
x = tf.Variable(3.0, trainable=True)
with tf.GradientTape(watch_accessed_variables=False) as tape:
    y = x**3

print(tape.gradient(y, x))
```

خروجی این تکه کد None است.

مثال ۴: محاسبه مشتق های مرتبه بالاتر



```python
x = tf.Variable(3.0, trainable=True)
with tf.GradientTape() as tape1:
    with tf.GradientTape() as tape2:
        y = x ** 3
    order_1 = tape2.gradient(y, x)
order_2 = tape1.gradient(order_1, x)

print(order_2.numpy())  # -> 18.0
```

برای این کار باید از tf.GradientTape به صورت تودرتو استفاده کنیم.

مثال ۵ : چندین بار استفاده از gradient

```
a = tf.Variable(6.0, trainable=True)
b = tf.Variable(2.0, trainable=True)
with tf.GradientTape() as tape:
    y1 = a ** 2
    y2 = b ** 3

print(tape.gradient(y1, a).numpy())
print(tape.gradient(y2, b).numpy())
```

```
RuntimeError: A non-persistent GradientTape can only be used to compute one set of gradients
```

نحوه حل :استفاده از persistent

```
a = tf.Variable(6.0, trainable=True)
b = tf.Variable(2.0, trainable=True)
with tf.GradientTape(persistent=True) as tape:
    y1 = a ** 2
    y2 = b ** 3

print(tape.gradient(y1, a).numpy())
print(tape.gradient(y2, b).numpy())
```